2INC0 - Operating Systems

Deadlocks

Geoffrey Nelissen

IRiS Interconnected Resource-aware Intelligent Systems

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Course Overview

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduling** (lectures 2+3)
- **Concurrency and synchronization**
  - atomicity and interference (lecture 4)
  - actions synchronization (lecture 5)
  - condition synchronization (lecture 6)
  - **deadlock (lecture 7)**
- **File systems** (lecture 8)
- **Memory management** (lectures
- **Input/output** (lecture 11)

- Several solution to synchronize the execution of tasks and prevent unwanted traces
  - **Mutual exclusion**
  - **Action synchronization** (using semaphores)
  - **Condition variables**
  - **Monitors**

- How to analyze, avoid, detect, recover from **deadlocks**

Agenda

- Reminder
- Analysis of deadlocks
- Dealing with deadlocks

Deadlocks definitions

Blocked

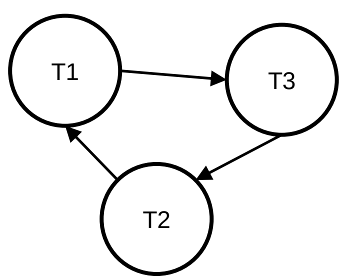
- A task is blocked if it is **waiting** on a blocking **synchronization action** (with another task, I/O device or other external component)

Deadlock

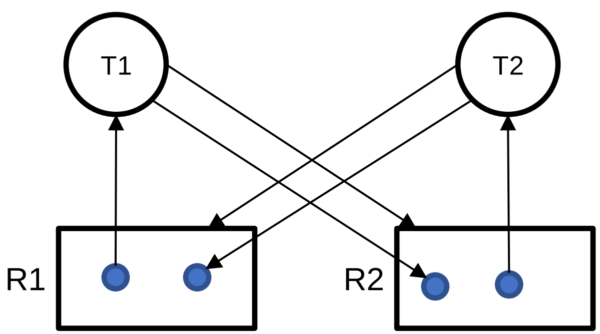
- When a subset of tasks is blocked indefinitely and cannot make progress anymore

Deadlocked set

- A set of task *D* is deadlocked if
  - **all** tasks in *D* are **blocked or terminated** (normally or abnormally),
  - there is at least **one non-terminated task** in *D*, and
  - for each non-terminated task *T* in *D*, any **task that might unblock *T* is also in *D*.**



Wait-for graph



Dependency graph

Resource types

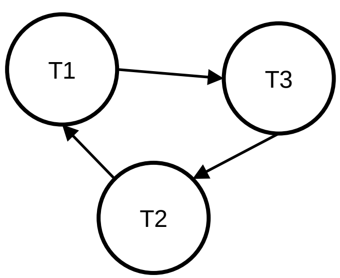
TU/e

Consumable resources

- **resources is taken away** upon use (➔ number of resources varies)
  - Typical **producer / consumer** problems
  - **examples**: sensor data, characters typed using a keyboard, blocks of data received from the network, …

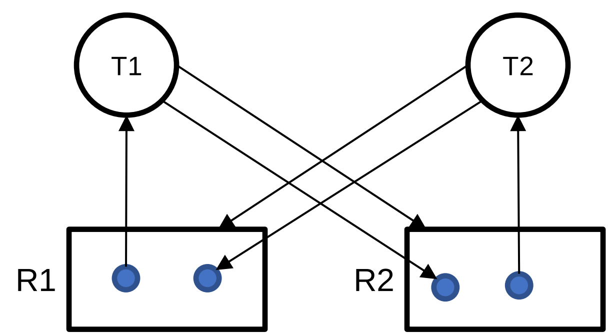
Reusable resources

- **resources are given back** after use (➔ number of resources is fixed)
  - Typically, **readers/writers** type of problems or **mutual exclusion** (critical section)
  - **examples**: processor, memory blocks, shared variables, buffer spaces, …



Wait-for graph

Analysis of consumable resources and condition variables



Dependency graph

Analysis of reusable resources and actions synchronization

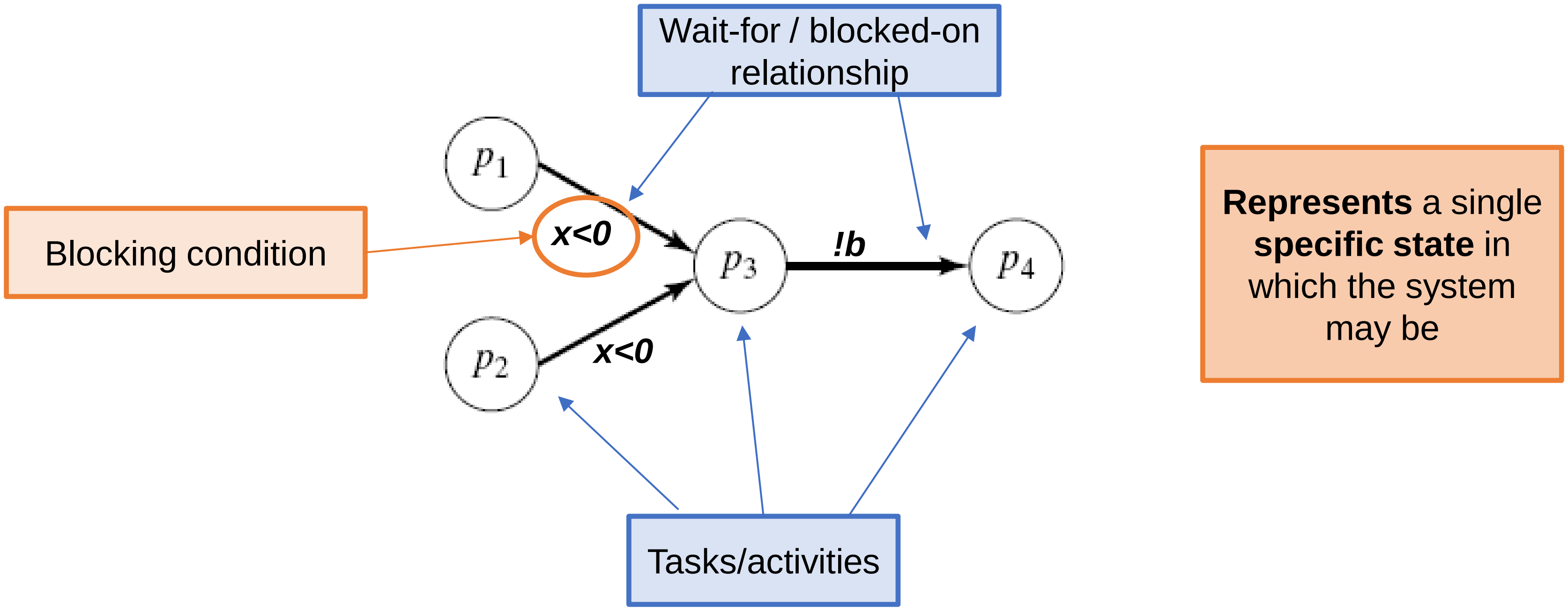
IRiS

Agenda

- Reminder
- Analysis of deadlocks
  - Wait-for graphs
  - Dependency graphs
- Dealing with deadlocks

Wait-for graph

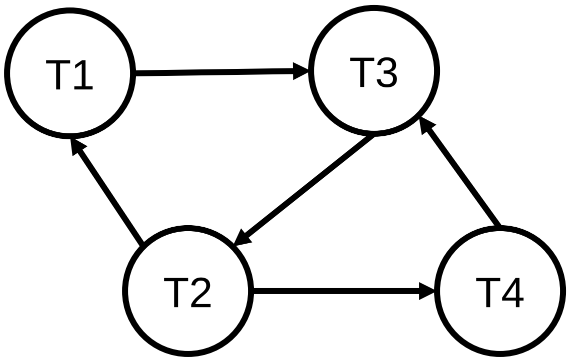
- Used to analyze **consumable resources** and **condition synchronizations**
- An edge ***p1* → *p3* means that *p3 is blocked and p1* may unblock *p3 by <u>falsifying the blocking condition</u>***



The wait-for graph represents a **deadlock state** if and only if

- there is a **cycle in the graph and**
- **no task outside any cycle** can unblock a task in the cycle

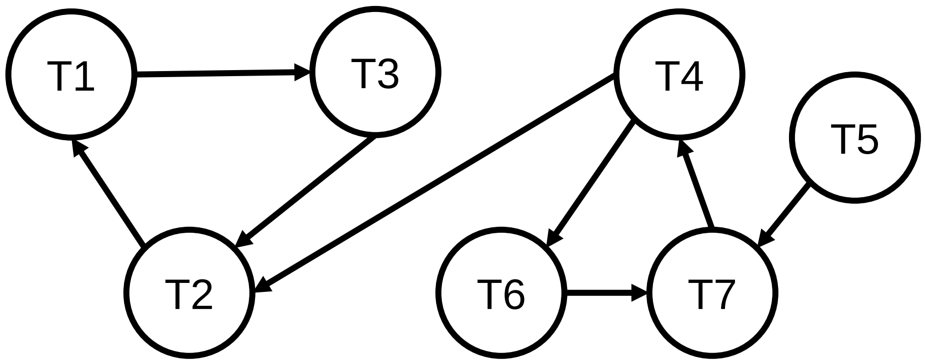
Examples



Is it in a **deadlock state**?

yes

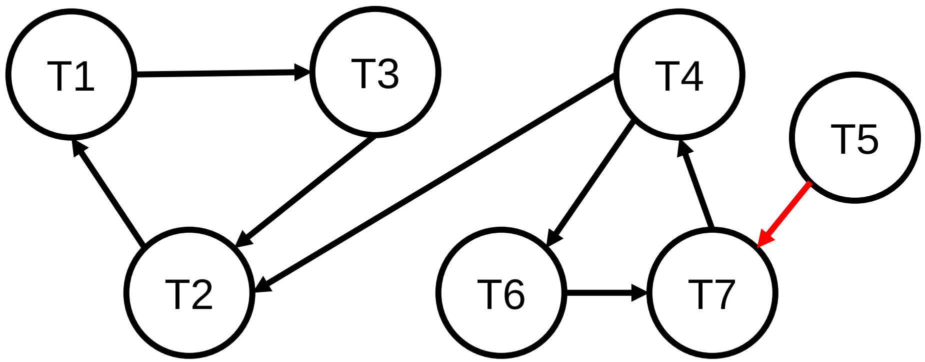
Examples



Is it in a **deadlock state**?

no

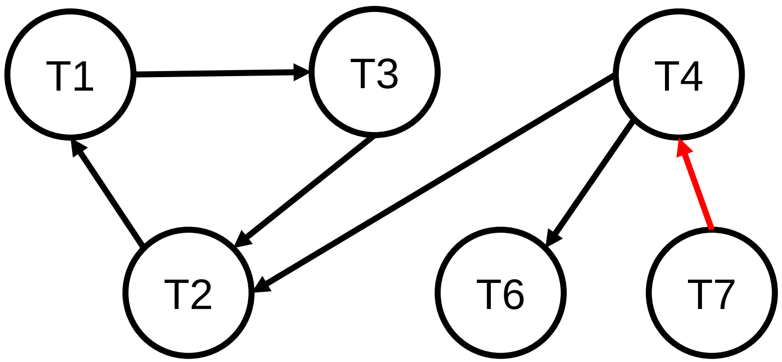
Examples



Is it in a **deadlock state**?

no

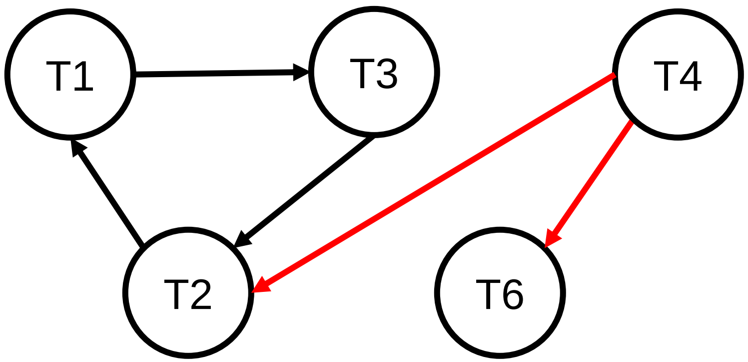
Examples



Is it in a **deadlock state**?

no

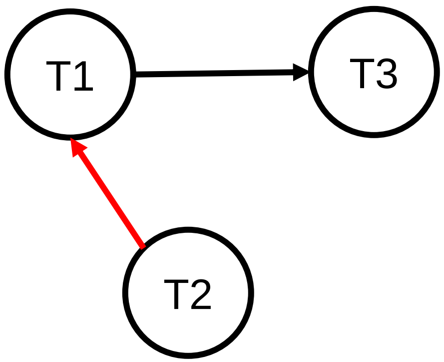
Examples



Is it in a **deadlock state**?

no

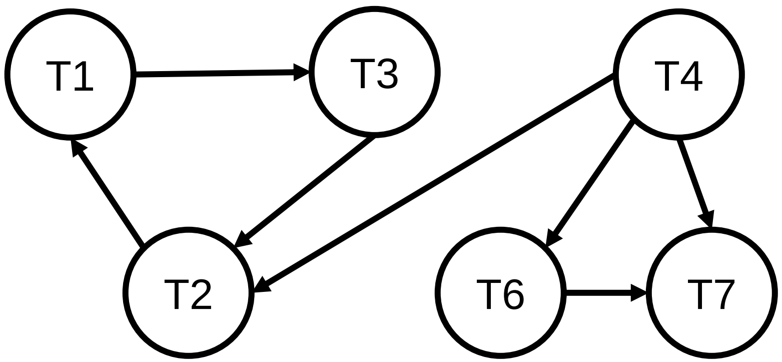
Examples



Is it in a **deadlock state**?

no

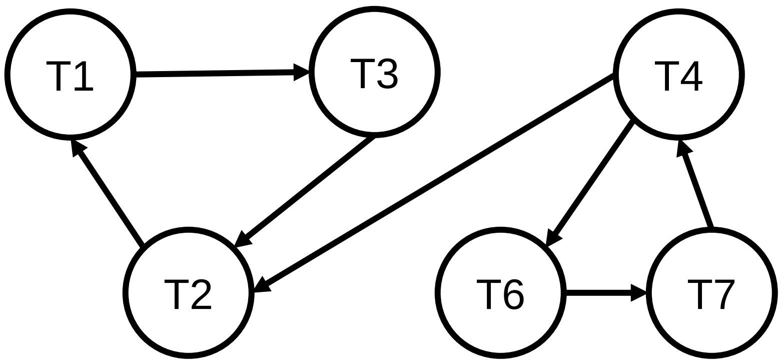
Examples



Is it in a **deadlock state**?

no

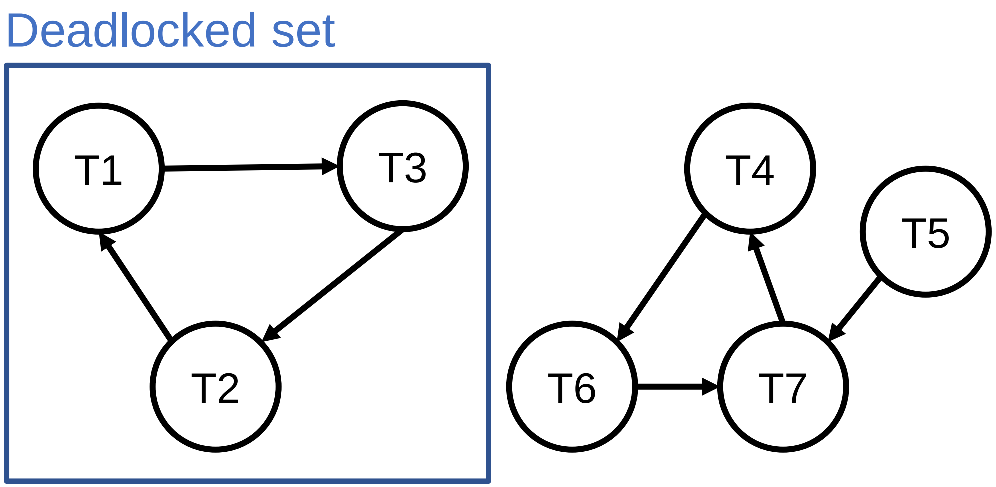
Examples



Is it in a **deadlock state**?

yes

Examples



Is it in a **deadlock state**?

Yes

How to use a wait-for graph to prove the absence of deadlocks?

- Use a **proof by contradiction**
  1. **Assume** one task ***T* is blocked** on a certain action
     - **Add *T* to the graph**
  2. Check **which** other **task(s) can unblock *T*** and *under which condition*
     - **Add each task *T'*** that can unblock *T* **to the graph**
     - **create an edge from *T'* to *T*** labeled with the blocking condition it may unblock
  3. Check whether each task ***T'* can possibly be blocked** on an action
     - If yes, **repeat from 1** with *T'*
  4. Once the whole graph is built check if we are in a deadlock state (i.e., there is an unbreakable cycle)
     - If **no**, we reached a **contradiction**
     - If **yes**, we are in a potential **deadlock** and we should **build a trace** showing how we reached that state

Build a wait-for graph for **every** action
that may block a task

Exercise

TU/e

```
T1 = |[
    while (true) {
        P(m);
        x := x+y;
        sigall(c);
        V(m);
    }
]|
```

||

```
T2 = |[
    while (true) {
        P(m);
        x := a;
        sigall(c);
        V(m);
    }
]|
```

||

```
T3 = |[
    while (true) {
        P(m);
        while (x<0) {
            wait(m,c);
        }
        x := x-1;
        b := true;
        signal(d);
        V(m);
    }
]|
```

||

Blocked

```
T4 = |[
    while (true) {
        P(m);
        while (!b) {
            wait(m,d);
        }
        b := false;
        V(m);
    }
]|
```

Can we possibly be in a deadlock state when T4 is blocked on *wait(m,d);*?
Build the wait-for graph.
Assume there is only one instance of each task.

IRiS

Example

```
T1 = |[
    while (true) {
        P(m);
        x := x+y;
        sigall(c);
        V(m);
    }
]|
```

May unblock T3

||

Blocked

```
T2 = |[
    while (true) {
        P(m);
        x := a;
        sigall(c);
        V(m);
    }
]|
```

May unblock T3 if a>0

||

Blocked

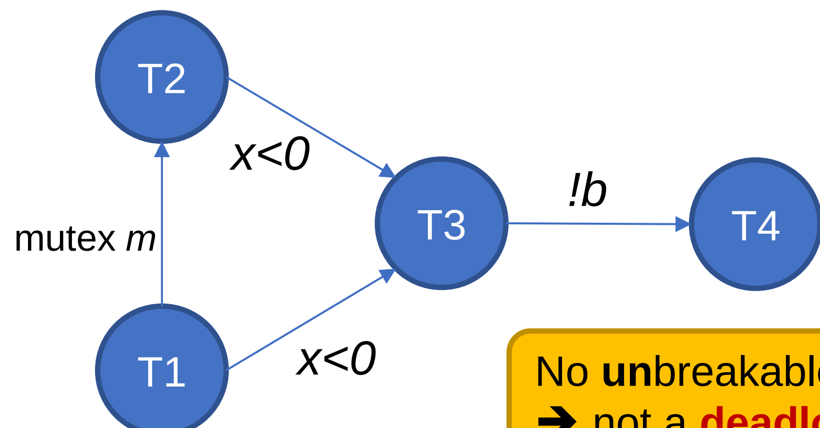
```
T3 = |[
    while (true) {
        P(m);
        while (x<0) {
            wait(m,c);
        }
        x := x-1;
        b := true;
        signal(d);
        V(m);
    }
]|
```

May unblock T4

||

Blocked

```
T4 = |[
    while (true) {
        P(m);
        while (!b) {
            wait(m,d);
        }
        b := false;
        V(m);
    }
]|
```



No **un**breakable cycle
➔ not a **deadlock state**

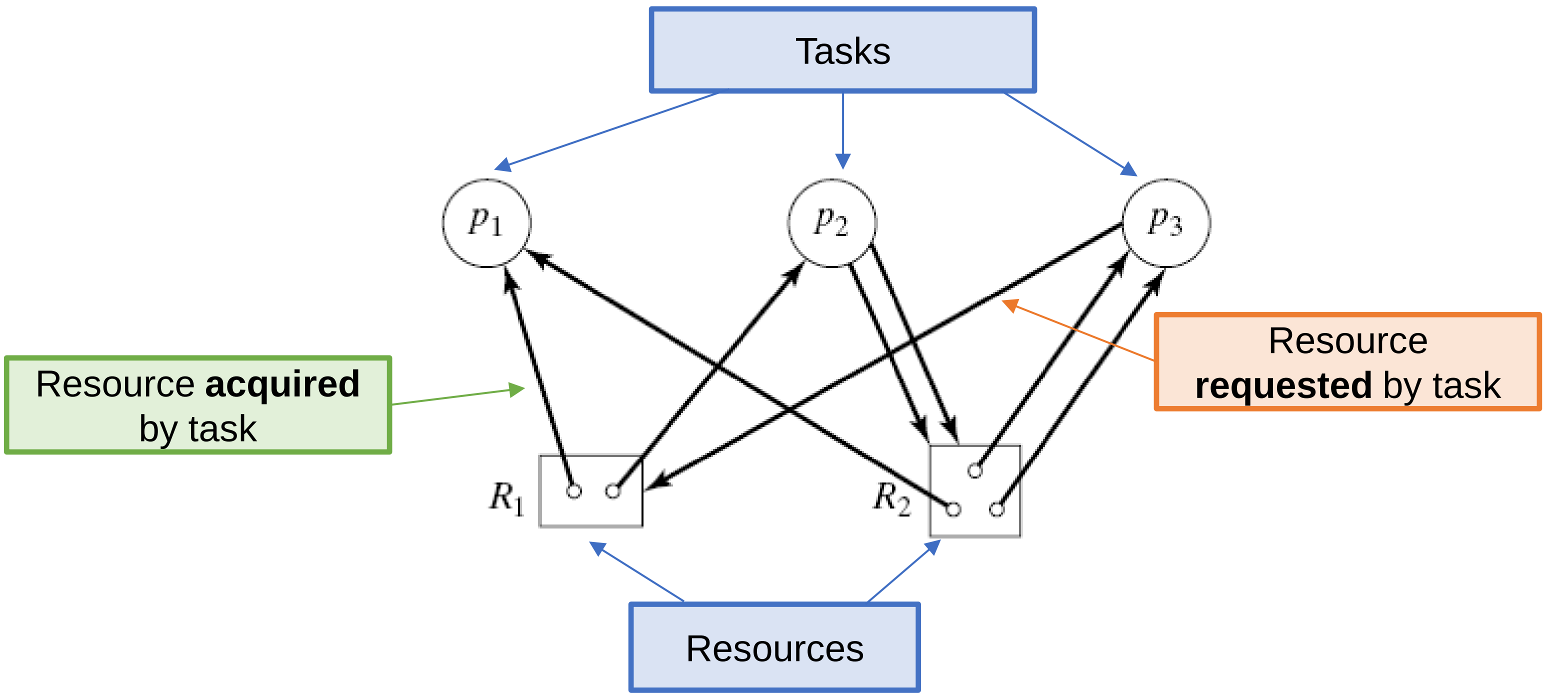
Must do the same for every blocking action to prove that the program cannot deadlock

Agenda

- Reminder
- Analysis of deadlocks
  - Wait-for graphs
  - Dependency graphs
- Dealing with deadlocks

Dependency graph

- Used to analyze **reusable resources** (and actions synchronization)

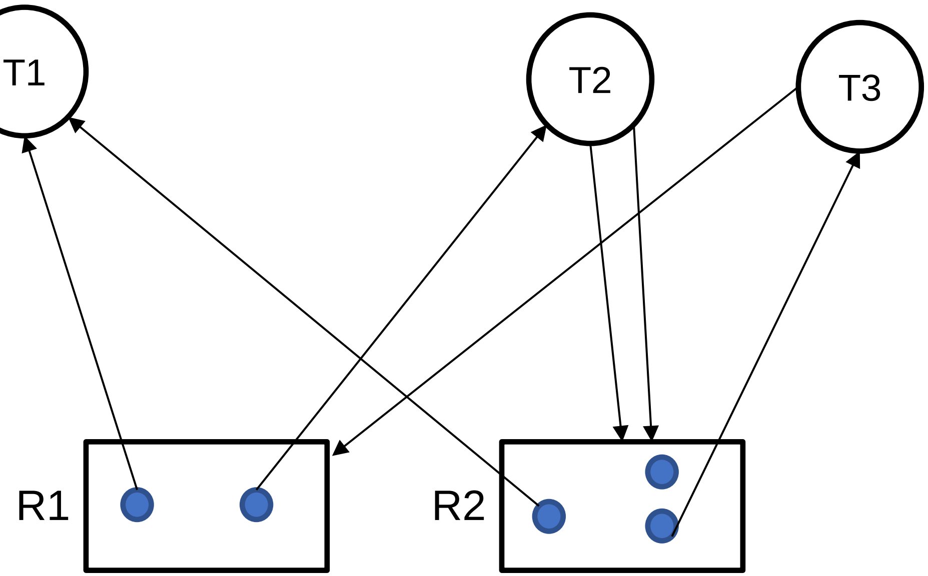


A **task is blocked** if it has an outgoing edge that is not directly removable, i.e., for which the **requested resource(s) are not free**

Reduction of dependency graph: **repeatedly remove all non-blocked tasks**

The dependency graph represents a **deadlock state** if the **reduced graph is not empty**

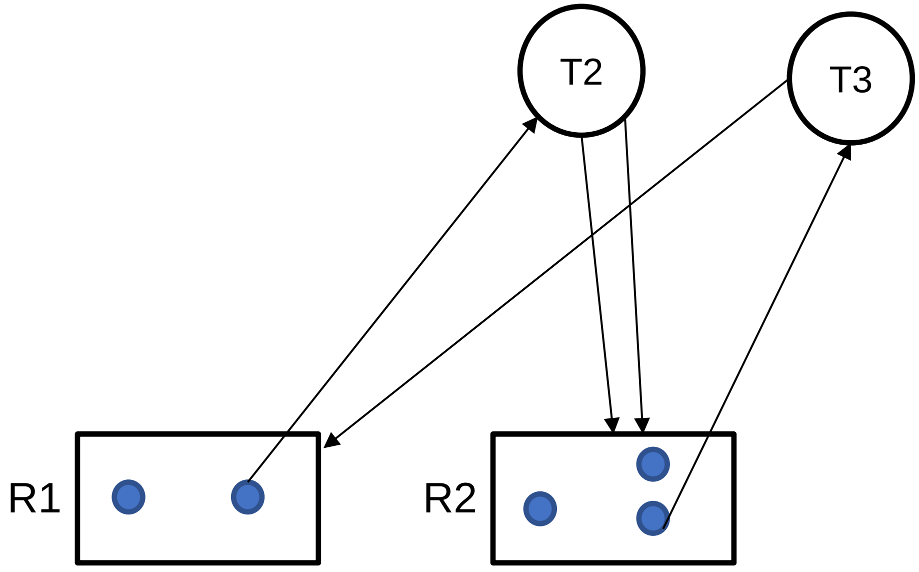
Example



Are we in **a deadlock**?

no

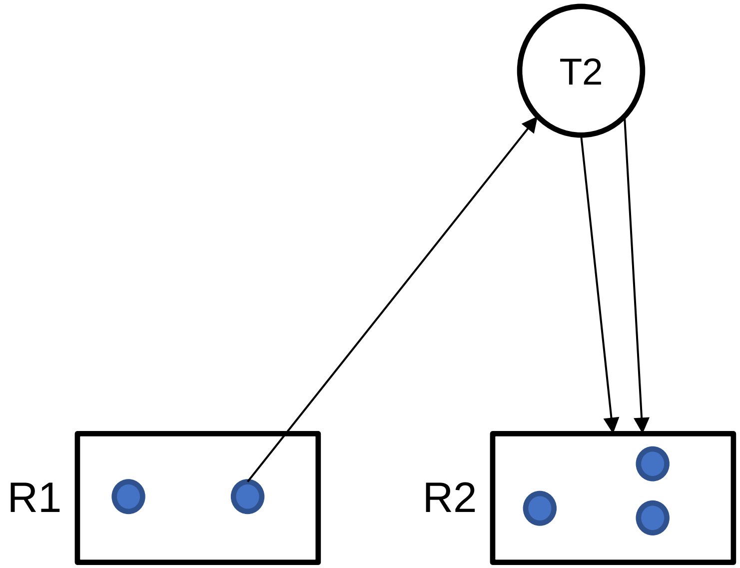
Example



Are we in **a deadlock**?

no

Example

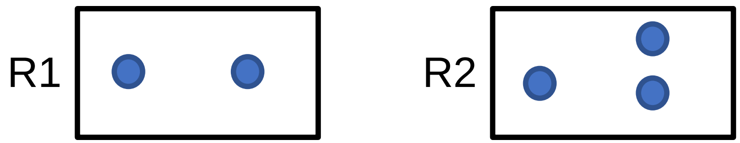


Are we in **a deadlock**?

no

Example

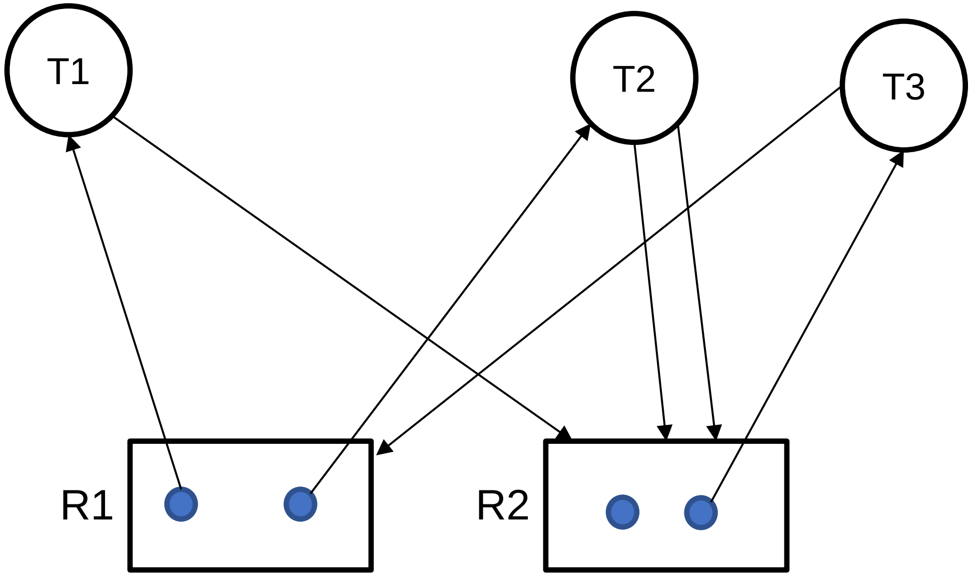
Note: the reduction assumes that all tasks **always** eventually **release** all the **resources** they have acquired



Are we in **a deadlock**?

No, the reduced graph is empty

Example



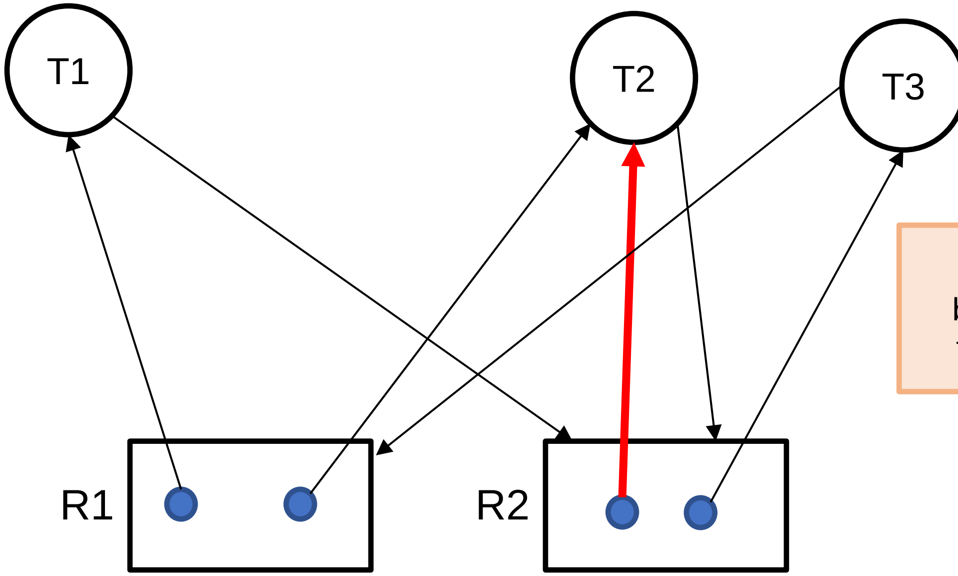
Are we in a **deadlock**?

no

Can we **reach a deadlock** from this state?

What tasks would be in the **deadlocked set**?

Example



If T2 acquires R2 before T1, then all tasks are blocked

Are we in a **deadlock**?

no

Can we **reach a deadlock** from this state?

yes

What tasks would be in the **deadlocked set**?

T1, T2 and T3

Using resource dependency graphs to show the absence of a deadlock state

- **We must show there is no knot** (i.e., the graphs are reducible to an empty graph) **in all the dependency graphs that can be generated by allocating the resources according to the program code**

- **More generally,** examine all the reachable states of the **Finite State Machine** corresponding to the request/acquisition sequences

Exercise 2

```
semaphore m1 := 1;
semaphore m2 := 1;
semaphore m3 := 1;
```

```
Task T1 =
|[ while( true ){
        P(m1);
        P(m2);
        doSomething();
        V(m2);
        V(m1);
}
]|
```

```
Task T2 =
|[ while( true ){
        P(m2);
        P(m3);
        doSomething();
        V(m3);
        V(m2);
}
]|
```

```
Task T3 =
|[ while( true ){
        P(m3);
        P(m1);
        doSomething();
        V(m1);
        V(m3);
}
]|
```

Draw the dependency graph of a reachable state in which that system is in a deadlock

Exercise 2: solution

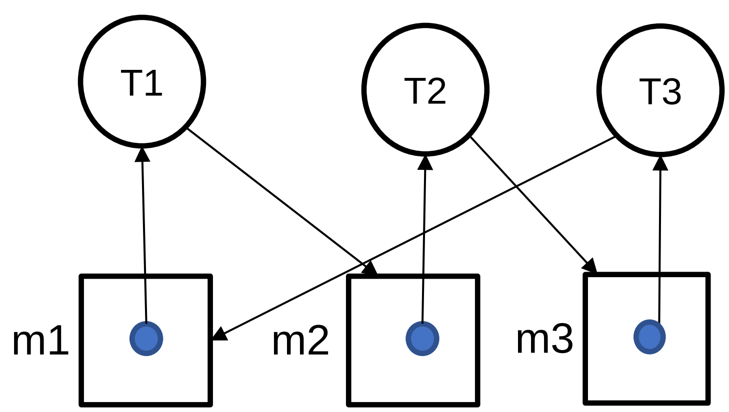
```
semaphore m1 := 1;
semaphore m2 := 1;
semaphore m3 := 1;
```

```
Task T1 =
|[ while( true ){
        P(m1);
        P(m2);
        doSomething();
        V(m2);
        V(m1);
}
]|
```

```
Task T2 =
|[ while( true ){
        P(m2);
        P(m3);
        doSomething();
        V(m3);
        V(m2);
}
]|
```

```
Task T3 =
|[ while( true ){
        P(m3);
        P(m1);
        doSomething();
        V(m1);
        V(m3);
}
]|
```

Simply show we can reach this state that is not reducible to an empty graph



Agenda

- **Reminder**
- **Analysis of deadlocks**
  - **Wait-for graphs**
  - **Dependency graphs**
- **Dealing with deadlocks**

Dealing with deadlocks

- It is **often ignored** (since assumed to be rare) at design time.
- However, it may have **catastrophic results** if it indeed happens
  - **Examples**: control system, health device, auto-pilot, buying-selling stocks, ...

- Three active approaches to dealing with deadlocks:
  - **Prevention** (programmer side)
    - at design time
  - **Avoidance** (system side)
    - dynamicly checks to avoid entering risky states (can be expensive)
  - **Detection and recovery**
    - checks only whether we are in a deadlock state, and try to recover from it

Dealing with deadlocks

- It is **often ignored** (since assumed to be rare) at design time.
- However, it may have **catastrophic results** if it indeed happens
  - **Examples**: control system, health device, auto-pilot, buying-selling stocks, ...

- Three active approaches to dealing with deadlocks:
  - **Prevention** (programmer side)
    - at design time
  - **Avoidance** (system side)
    - dynamicly checks to avoid entering risky states (can be expensive)
  - **Detection and recovery**
    - checks only whether we are in a deadlock state or not, and try to recover from it

Prevention (at design time)

- **Analyse** your system using ***reduced dependency graphs*** and ***wait-for graphs*** at design time (i.e., **provide a proof** that there can never be any deadlock)
  - make the **reduced dependency graphs empty by construction**
  - **prevent cycles in the wait-for graphs**

- Use **synchronization** tricks (see "actions synchronization")
  - Examples:
    - **no circular wait,**
    - **ensure terminating critical sections**
    - **Lock resources in a fixed order**
    - Acquire **"all resources at once", i.e., avoids the "wait-and-hold" greediness**
  - not always possible

- Allow **preemption of resources** when needed
  - **Examples**: add **timeout** on how long a task can hold a resource, or allow a task to **steal a resource** from another one
  - not always possible (e.g., an I/O device may be in an **inconsistent state** if preempted during an I/O operation)

Dealing with deadlocks

- It is **often ignored** (since assumed to be rare) at design time.
- However, it may have **catastrophic results** if it indeed happens
  - **Examples**: control system, health device, auto-pilot, buying-selling stocks, ...

- Three active approaches to dealing with deadlocks:
  - **Prevention** (programmer side)
    - at design time
  - **Avoidance** (system side)
    - dynamicly checks to avoid entering risky states (can be expensive)
  - **Detection and recovery**
    - checks only whether we are in a deadlock state or not, and try to recover from it

Avoidance (during execution)

- Upon **executing** each potentially **blocking action** (e.g., *P(m)*):
  - **check if an open execution path exists** (i.e., there remains a sequence of possible actions such that we will avoid deadlocks, that is, **the system remains** in a **"safe"** state)
  - **otherwise, deny or postpone** the action

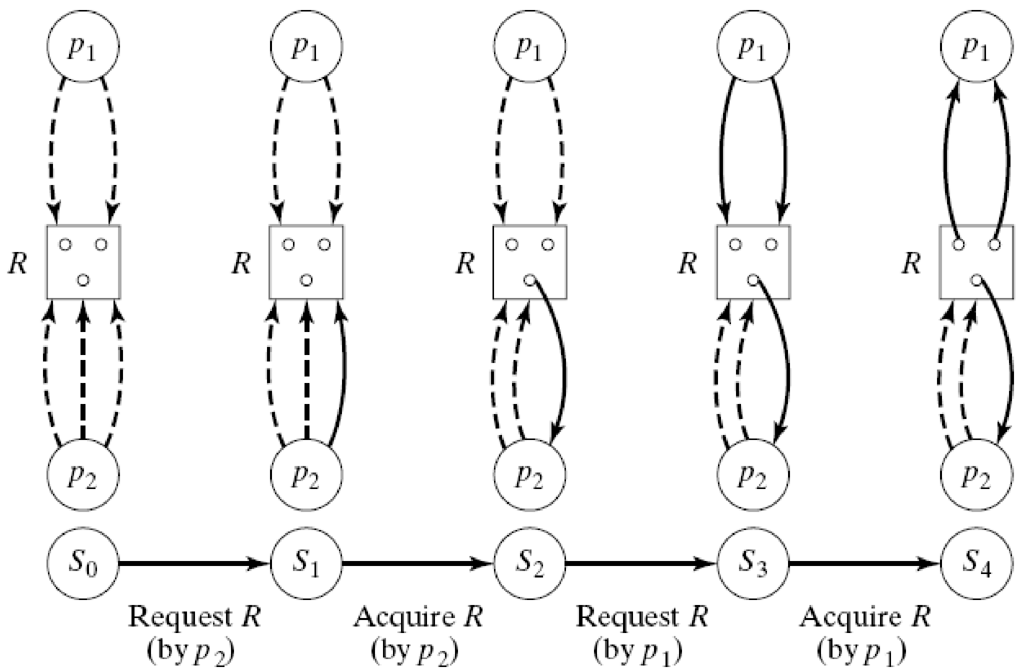
- How to check the existence of an **open path**? ➔ the **reduced max claim graph is empty** (see next slide)

- Avoidance by postponing actions **works well if** the blocking actions refer to allocations of **reusable resources** and **we can anticipate all possible future states**

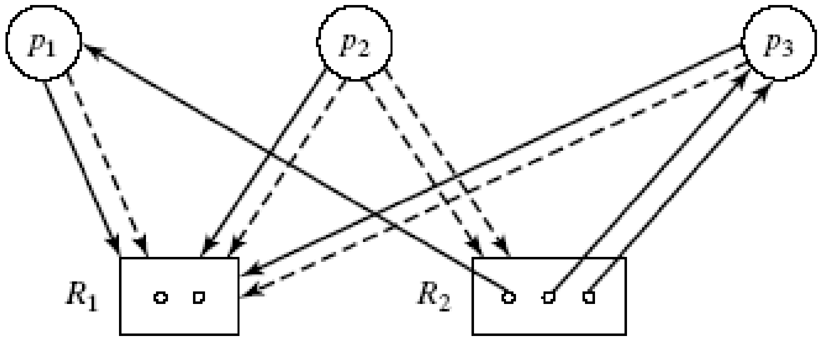
- **Example of such avoidance algorithm**: the **banker's algorithm**

(Maximum) claim graph

- **Max. claim graph = dependency graph extended** to capture potential future resource claims
- The (maximum) **future resource claim** is illustrated in the resource dependency graph, **using dashed arrows**
  - It is then called a ***maximum* claim graph**



Example: avoiding deadlock



(a) p_1 *and* p_2 *require* R_1

Should we allow a state transition

- ***p_1 acquires R_1?***
- Yes because it leaves a **reducible graph**

Should we allow a state transition

- ***p_2 acquires R_1?***
- No because it becomes a **non-reducible graph** – action should be denied or postponed

- Note: while in state (a), granting *p3's* request is also safe

Banker's algorithm: problem description

- Given a set of *N* tasks, set of *R* resource types with
  - $c[j]$: number of resources of type *j*
  - $\max[i, j]$: maximum number of resources of type j that **may** be needed by task T_i

- **Requirement**
  - synchronize requests such that each task can always acquire resources until its specified maximum
    - not satisfying this requirement **may** lead to a deadlock

- **Assumptions**: Tasks acquire resources incrementally (**greedy algorithm**) and **release those eventually**

Formalization

- **Inputs**

  Set of *N* tasks, set of *R* resource types

  - $c[j]$: number of resources of type *j*
  - $\max[i,j]$: maximum number of resources of type j that **may** be needed by task T_i

- **System state**

  represented by

  - $avail[j]$: number of resources of type *j* still available
  - $alloc[i,j]$: number of resources of type *j* already allocated to task T_i
  - $claim[i,j]$: maximum number of resources of type *j* that **may** still be claimed by task T_i

- **Initial state**

  - $avail[j] = c[j]$
  - $alloc[i,j] = 0$
  - $claim[i,j] = \max[i,j]$

- **Invariants**

  - $avail[j] = c[j] - \sum_i alloc[i,j]$
  - $claim[i,j] + alloc[i,j] = \max[i,j]$

Formalization (cnt'd)

- A **state is** called ***open* for that task *T_i*, if all the resources it may claim can be given directly** to that task , i.e, $\forall j,\ claim[i,j] \leq avail[j]$

- A **state is** called ***safe* for a task, if** this **task can be given its maximum number of resources eventually**
  - possibly by giving available resources to other tasks first and then waiting until these release them again

- A **state is** called **safe** if it is safe for all tasks.

- **Banker's algorithm**:
  - **We grant a request only if the resulting state is safe**
    - note: requesting does not change state safety
  - Assumes the initial state is safe

Check state safety upon new request

- Assume the **current state is safe**
- Consider a **new request $req[i,j]$ on resource of type *j* by task T_i**
- Compute the **new state** if the request is granted
  - $avail[j] = avail[j] - req[i,j];$
  - $alloc[i,j] = alloc[i,j] + req[i,j];$
  - $claim[i,j] = claim[i,j] - req[i,j];$

It is **enough** to check whether the **new state is safe for task T_i**
if it is, then all the resources owned by T_i will eventually be returned
thus eventually reaching a state at least as safe as the current state

Remember: the new state is **safe for task T_i if its claims can be satisfied** directly or by completing the claims of any of other tasks

Implication:
to check the state safety for task *Ti*
=
perform a reduction of the dependency graph

Algorithm to check state safety:

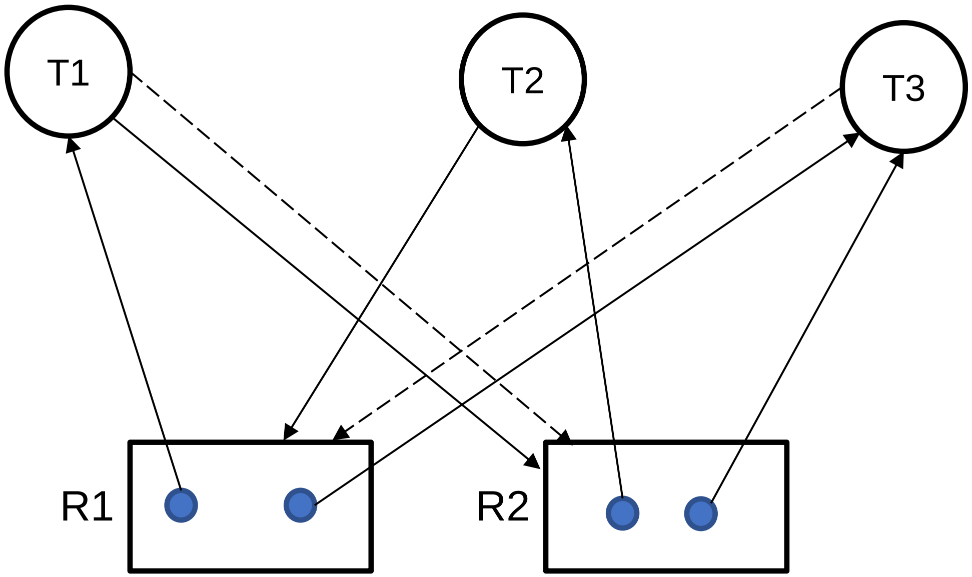
Repeat:
- find an open task
- remove it, and all its claims (= graph reduction)
- until task T_i is found to be open

Algorithm

```
int NextOpen (Vector Avail, Matrix Claim, Matrix Alloc)
// returns the index of first open task that has claimed resources
{
   for( i = 0 to N-1 ) {
      if (Alloc[i] != 0 and Claim[i] <= Avail) { /* Note that these tests manipulate vectors and make
                                                    element-wise comparisons */
            return(i);
      }
   }
   return(N); // if there is no open task returns N
};
```

```
bool Safe (Vector Avail, Matrix Claim, Matrix Alloc, Vector req, int i)
/* returns whether the transition results in a new safe state
  precondition: Av >= req (otherwise the transition is impossible anyway) */
{
   int k;
   Avail, Alloc[i], Claim[i] = Avail-req, Alloc[i]+req, Claim[i]-req; // compute the new state
   k = NextOpen (Avail, Claim, Alloc); // get the next open task
   while (k != N and not (Claim[i] <= Avail) ){
   // there is an open task and i is not open
         Avail, Alloc[k], Claim[k] = Avail+Alloc[k], 0, Max[k]; // assume open task k completes
         k = NextOpen (Avail, Claim, Alloc); // get the next open task
   }
   return (Claim[i] <= Avail );
}
```

Exercise

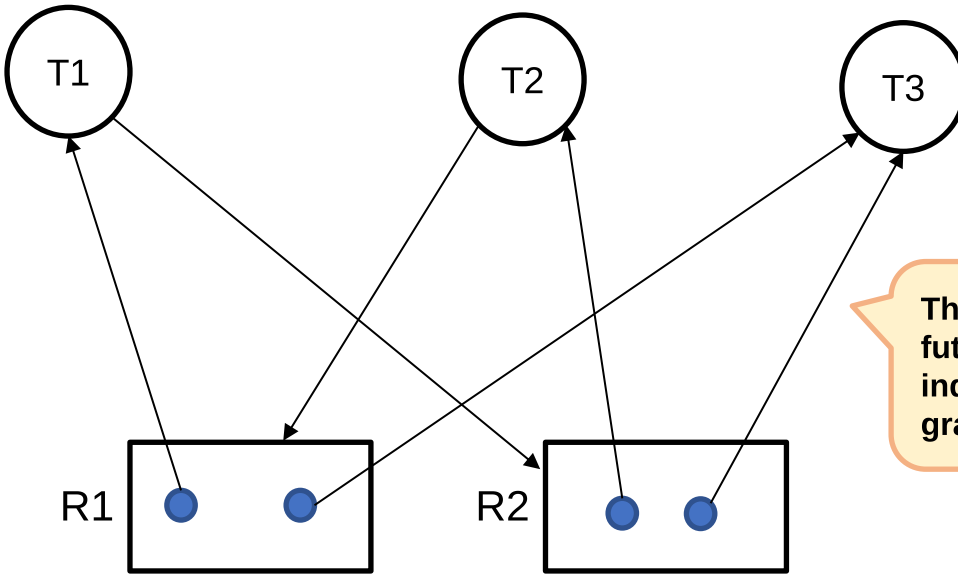


Is this state safe?

No, the reduced graph is not empty

Are we in a deadlock?

Exercise



The dependency graph without future potential requests is indeed reducible to an empty graph

Is this state safe?

No, the reduced graph is not empty

Are we in a deadlock?

No

Remember: you check whether you are in a deadlock by analyzing the **dependency graph without accounting for future potential requests, i.e., without dashed edges**

Dealing with deadlocks

- It is **often ignored** (since assumed to be rare) at design time.
- However, it may have **catastrophic results** if it indeed happens
  - **Examples**: control system, health device, auto-pilot, buying-selling stocks, ...

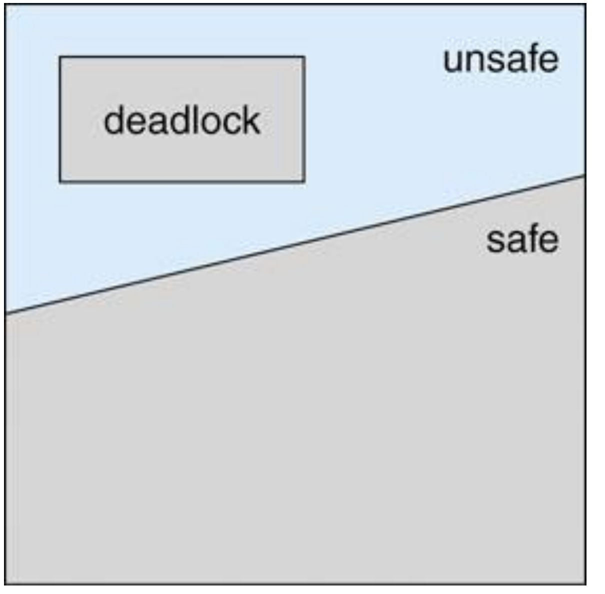
- Three active approaches to dealing with deadlocks:
  - **Prevention** (programmer side)
    - at design time
  - **Avoidance** (system side)
    - dynamicly checks to avoid entering risky states (can be expensive)
  - **Detection and recovery**
    - checks only whether we are in a deadlock state or not, and try to recover from it

Detection (during execution)

- Invoke detection algorithm periodically to check if deadlock occurs

- **You need**
  - an algorithm to **examine the state upon execution of a blocking action**
  - an **algorithm to recover** from a deadlock

- Repeatedly monitoring and potentially recovering causes large **overheads**



Detection (during execution)

- **Recovery algorithm** may be executed
  - **locally**, inside the task that last blocked (e.g., release all its resources)
  - **globally**, through a recovery policy
    - **kill** tasks in the deadlock set
      - all
      - selectively, **based on some criteria** (e.g., priority, progress made, etc.)
    - **roll back** to safe state
      - works only if an alternative execution path exists
      - Requires to record the state at *checkpoints*, i.e., keep enough history information to roll back
    - **preempt resources,** i.e., we dealocate resources from tasks in the deadlock set (not always possible)
      - select victim based on criteria

Overview

TU/e

+ possible
starvation

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommit resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses

Source:
Isloor and Marsland, *"The Deadlock Problem: An Overview,"* in *Computer*, vol. 13, no. 9, pp. 58-78, Sept. 1980, doi: 10.1109/MC.1980.1653786.
Stallings, William, and Goutam Kumar Paul. *Operating systems: internals and design principles*. Vol. 9. New York: Pearson, 2012.

IRiS

Summary

- We discussed **how to analyze deadlocks** using
  - **Wait-for graphs** for consumable resources
  - **Dependency graphs** for re-usable resources
- We discussed how to **avoid** deadlocks using the banker's algorithm
- Deadlock detection

- During the next lecture, Mitra Nasri will discuss **file systems**
- **Workshop on mutex, semaphores, threads, … on Friday**

- In two weeks, we will discuss **memory management** and **memory virtualization**
- **Preparation for memory management:**
  - **More material than usual ➔ start preparing earlier**